Appendice D

Architettura EPIC

Giacomo Bucci

Revisione del 31 marzo 2017

Questo documento è una appendice al volume
Calcolatori Elettronici - Architettura e Organizzazione
IV edizione
McGraw-Hill Education (Italy), S.r.l.
Milano, 2017

Storia degli aggiornamenti

Marzo 2017: primo rilascio.

D

L'architettura EPIC

OBIETTIVI

- Presentare l'architettura EPIC
- Illustrare le caratteristiche salienti del processore Itanium
- Illustrare l'esecuzione predicativa (ovvero condizionale)

CONCETTI CHIAVE

Architettura VLIW, parallelismo "statico", impaccamento delle istruzioni, esecuzione predicativa, load speculativo.

INTRODUZIONE

Nel 1989 i progettisti HP arrivarono a stabilire che le architetture RISC stavano raggiungendo il limite delle prestazioni da esse ottenibili e che ulteriori avanzamenti sarebbero stati possibili con l'adozione di architetture VLIW, i cui principi di funzionamento erano stati definiti sin dagli anni '80 [Fis83]. Venne lanciato un progetto congiunto HP-Intel per la definizione di una nuova, moderna architettura VLIW, denominata EPIC (*Explicit Parallel Instruction Computer*) [Dul98].

Il primo processore della famiglia, denominato Itanium [HMR$^+$00], apparve nel 2001. Tuttavia, pochi mesi dopo venne introdotto l'Itanium 2, versione migliorata del precedente, resasi necessaria dalle sue non brillantissime prestazioni [MS03], [RMC04].

Tra le caratteristiche dell'Itanium c'è quella di poter eseguire codice $\times 86$ a 32 bit.

Nelle intenzioni delle due compagnie, l'EPIC avrebbe dovuto prendere il posto della precedente architettura $\times 86$, nella versione di allora a 32 bit. Ma nel lungo tempo le aspettative sono andate deluse e la dominanza del mercato è ancora mantenuta dall'architettura $\times 86$, nella versione a 64 bit.

L'Itanium, sebbene sulla carta sia una macchina con eccellenti soluzioni tecniche, non è riuscito ad affermarsi. Come del resto è capitato ad altre architetture potenzialmente di grande pregio, ma che non hanno potuto intaccare la diffusione della $\times 86$.

Questa appendice introduce l'esecuzione predicativa, un modo di operare della logica di macchina in base al quale un'istruzione, per cui esiste un *predicato*, cioè una condizione, viene eseguita solo se il predicato è vero. L'esecuzione speculativa è una caratteristica anche dell'architettura ARM descritta nell'Appendice E. Per i concetti di carattere generale circa le architetture VLIW si faccia riferimento al Capitolo 10.

D.1 L'architettura EPIC

L'architettura EPIC proponeva caratteristiche innovative, [ZRH00][RGB+02], tra cui:

1. Il parallelismo esplicito, ovvero l'identificazione da parte del compilatore di gruppi di istruzioni contigue indipendenti, che possono essere eseguite in parallelo dalla logica di macchina.
2. L'esecuzione predicativa, ovvero l'esecuzione condizionale delle istruzioni, in grado di evitare la penalizzazione delle diramazioni mal predette.
3. Un ampio insieme di registri fisici, organizzati a stack per rendere efficiente il passaggio dei parametri.
4. La speculazione di controllo, ovvero un meccanismo che consente di anticipare le istruzioni di load a precedere quelle di diramazione, al fine di annullare i ritardi dovuti alla loro latenza.
5. La speculazione sui dati, ovvero un meccanismo che consente al compilatore di spostare le istruzioni di load e quelle da esse influenzate a precedere le istruzioni di store potenzialmente conflittuali.

Nel seguito ci si riferisce alle caratteristiche generali dell'architettura, ma quando si entra nel dettaglio il riferimento è a l'Itanium 2 [MS03], [RMC04], anche se per brevità si usa la sola parola "Itanium".

Lo schema semplificato dell'Itanium è in Figura D.1. La macchina presenta 11 unità funzionali, 3 preposte alla gestione dei salti (B), 4 per scambio dati con la memoria (M), 2 di elaborazione sugli interi (I) e 2 per l'elaborazione in virgola mobile (F).

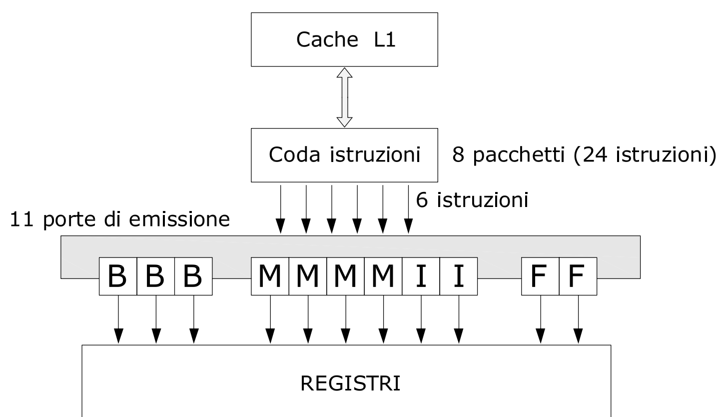


Figura D.1 Architettura del processore Itanium 2.

Poiché le unità M possono eseguire anche operazioni sugli interi, il processore è potenzialmente in grado di eseguire fino a 6 operazioni tra interi per ciclo di clock. Le istruzioni vengono caricate da una cache istruzioni di primo livello (L1) e poste in un buffer in grado di contenere fino a 8 pacchetti di istruzioni, per un totale di 24 istruzioni. Ad ogni ciclo di clock la CPU è potenzialmente in grado di trattare 2 pacchetti, ovvero 6 istruzioni.

Pacchetti e gruppi di istruzioni

La Figura D.2 mostra il formato di un pacchetto (*bundle*) di istruzioni Itanium. Un pacchetto occupa 128 bit e comprende tre istruzioni di 41 bit. Il repertorio prevede 6 tipi di istruzioni, eseguibili sulle differenti unità esecutive, come specificato in Tabella D.1. Alcuni tipi di istruzioni posso essere eseguiti su più di un tipo di unità esecutiva.

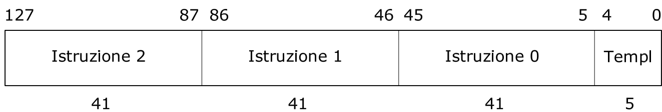


Figura D.2 Formato di un pacchetto (*bundle*) di istruzioni del processore Itanium. Il campo Templ (*template*) indica il tipo di ciascuna istruzione nel pacchetto (ovvero la mappatura di ogni istruzione su una unità di esecuzione), oltre a eventuali marchi di stop. L'ordinamento è little-endian.

Il campo di 5 bit Templ (*template*), presente in ogni pacchetto, indica ordinatamente quali tipi di istruzione esso contiene. Più precisamente, dei 32 possibili valori del campo Templ, 24 individuano la specifica combinazione dei tipi di istruzioni entro il pacchetto, mentre 8 sono riservati.

Tipo di istruzione	Descrizione	Tipo di unità esecutiva (UE)
A	Operazione ALU tra interi	I o M
I	Operazione tra interi non-ALU	I
M	Operazione di memoria	M
F	Operazione in virgola mobile	F
B	Operazione di salto	B
L+X	Operazione estesa	I o B

Tabella D.1 Tipi di istruzioni e unità funzionali su cui vengono eseguite. Le istruzioni di tipo A possono essere eseguite sia su unità I che M; le istruzioni di tipo F possono essere eseguite solo su F.

L'architettura Itanium dà vita a un ambiente di esecuzione parallela in cui il processore non deve esaminare le dipendenze tra i registri, né tantomeno deve garantire il corretto ordine delle istruzioni ritirate. Per ottenere un tale risultato, il compilatore organizza le istruzioni in gruppi. Un *instruction group* è una sequenza di istruzioni contigue indipendenti fra loro, che:

a) non confliggono nell'uso delle unità funzionali;
b) non confliggono nell'uso dei registri;
c) non hanno dipendenze tra loro.

Ogni gruppo può contenere un numero arbitrario di istruzioni, ma il compilatore deve indicare esplicitamente il limite fra un gruppo e il successivo, attraverso uno speciale marchio detto *stop*[1] . Le istruzioni di un gruppo vengono eseguite in parallelo; se per

[1] In realtà ci sono altre condizioni che fanno terminare un gruppo, che evitiamo di illustrare per non appesantire la trattazione. Accenniamo solo al fatto che un gruppo termina anche quando c'è un'istruzione di salto; se questo è condizionato, esso ha l'effetto di far terminare il gruppo solamente se risulta un salto

esempio un gruppo contiene più di 6 istruzioni a partire dalla posizione 0 entro un pacchetto, ne verranno eseguite 6 al primo clock e il resto al clock o ai clock successivi. Gruppi di istruzioni consecutivi vengono invece eseguiti in maniera sequenziale. Se la cosa non crea conflitti, l'esecuzione di un gruppo può iniziare prima che tutte le istruzioni del precedente siano terminate.

Gli stop e la loro posizione sono essi pure indicati attraverso il campo Templ. In Tabella D.2 si mostra un esempio di sequenza di pacchetti e di gruppi. Il valore 00 del *template* identifica un pacchetto di formato M-I-I; il valore 0A identifica ancora un gruppo M-I-I, ma con uno stop (indicato con un "*") tra le istruzioni M e I; il valore 03 identifica un pure un pacchetto di formato M-I-I, ma con lo stop posizionato tra le due istruzioni di tipo I. Il terzo e quarto pacchetto non presentano stop. Si noti che il template è di volta in volta differente, specificando esattamente i differenti contenuti dei pacchetti.

Templ	Slot 0 UE	Slot 1 UE	Slot 2 UE
00	M	I	I
0A	M *	I	I
0C	M	F	I
10	M	I	B
03	M	I *	I

Tabella D.2 Tipi di istruzioni e unità funzionali su cui vengono eseguite. Il contenuto di Templ definisce la combinazione dei tipi di istruzioni che formano il pacchetto e la posizione di eventuali stop all'interno. Lo stop è indicato con "*". Si noti che per convenienza questa figura è stata disegnata rovesciando l'ordine rispetto a quello little-endian (si confronti con la Figura D.2).

Con riferimento alla Tabella D.2, assumendo che l'esecuzione inizi con il primo pacchetto, il primo gruppo è costituito dalle 3 istruzioni del primo pacchetto (M-I-I) e dall'istruzione M del secondo, tra loro necessariamente indipendenti, a formare un gruppo di 4 istruzioni. Dopo la quarta istruzione c'è uno stop che individua la fine del pacchetto; esso implica una dipendenza con la quinta istruzione. Il passaggio all'esecuzione del secondo gruppo, quello che inizia con l'istruzione I che segue lo stop nel secondo pacchetto, si ha solo a conclusione dell'esecuzione del primo gruppo. Quando ciò accade, la CPU procede all'esecuzione del secondo gruppo che termina con l'istruzione di tipo I in colonna centrale del quinto pacchetto. Tuttavia, all'interno del secondo gruppo c'è un'istruzione di salto (B); se il salto viene preso, l'esecuzione del gruppo termina con questa istruzione, altrimenti comprende anche le due successive. L'influenza della struttura dei pacchetti è stata esaminata in [LBT06].

D.1.1 Modello di programmazione

Descriviamo ora, seppure in maniera molto succinta, il modello di programmazione.

Registri di CPU

La macchina presenta i seguenti registri di CPU (vengono descritti solo quelli di interesse per la discussione successiva).

effettivo, ovvero se determina una modifica del program counter, con conseguente passaggio ad altro gruppo di istruzioni.

- 128 registri di uso generale da 64 bit (GR0-GR127). Di questi, i registri GR0-GR31 sono denominati "statici". Sostanzialmente sono i normali registri di uso generale. Il registro GR0 contiene sempre 0; un tentativo di scrittura in questo registro determina un *Illegal Operation fault*. I registri GR32-GR127 sono denominati "registri di stack"; essi vengono usati nelle chiamate ai sottoprogrammi, secondo uno schema simile a quello visto nel libro per le CPU Sun. In appoggio a questa funzionalità c'è uno speciale registro detto *Current Frame Marker* (CFM) che descrive lo stato dello stack (dimensione delle porzioni di stack in uso). A ciascun registro GPR è associato un bit, invisibile al programmatore, denominato NaT (*Not a Thing*). Questi bit servono a generare eccezioni ritardate quando viene tentata la lettura di un registro il cui contenuto è stato caricato su un percorso speculativo errato (si veda più avanti).
- 128 registri in virgola mobile da 82-bit (FR0-FR127). I registri da 0 a 31 sono detti "statici". FR0 e FR1 contengono sempre +0,0 e +1,0 rispettivamente. Un tentativo di scrittura in questi registri determina un *fault*. I registri FR32-FR127 sono detti "rotanti" e possono essere ridenominati[2] da programma al fine di accelerare i cicli (*loop*). Inoltre quando l'Itanium esegue istruzioni IA-32, usa i registri FR8-FR31 per le operazioni in virgola e per le operazioni multimediali. Anche a ciascun registro FR è associato un NaT.
- 64 registri "predicativi" da 1 bit (PR0-PR63). Essi contengono i risultati delle istruzioni di confronto (*compare*) e vengono usati per l'esecuzione condizionale delle istruzioni. I registri PR0-PR15 sono detti "statici" e possono anche essere usati nelle istruzioni di salto condizionato. PR0 contiene sempre 1. I registri PR16-PR63 sono detti "rotanti" e possono essere ridenominati da programma per velocizzare i cicli.
- Altri registri, tra cui 8 registri (BR0-BR7) usati ai fini delle operazioni di salto e 128 registri applicativi (AR0-AR127), contenenti dati e informazioni di controllo. Questi registri possono essere visti dalle applicazioni (anche quelle IA-32). Per esempio AR24 rappresenta il registro EFLAG dell'architettura IA-32, mentre AR44 è un *interval timer*.

Formato delle istruzioni

Dal punto di vista della struttura delle istruzioni Itanium è sostanzialmente una macchina RISC, come illustrato in Figura D.3. Il campo OP (4 bit) contiene il codice principale di operazione, mentre il campo PR identifica il registro predicativo coinvolto.

Chiamata/ritorno alle/dalle procedure

I registri statici GR0-GR31 sono visibili a tutte le procedure. I registri di stack sono invece locali alle procedure; una procedura potrebbe anche impiegarli tutti quanti. All'atto della chiamata della procedura B da parte della procedura A, i registri di input della procedura B sono sovrapposti a quelli di output della procedura A, come schematizzato in Figura D.4. Ciò consente di evitare il passaggio esplicito dei parametri, ovvero la copiatura e il ripristino dei registri. Ogni procedura è responsabile dell'allocazione dello *stack frame* consistente nei parametri di ingresso, le variabili locali e i parametri di uscita. Un esame dei vantaggi dello stack dei registri di Itanium si trova in [RGB$^+$02]. Lo stesso hardware rinomina i registri usati dalla procedura in modo che essi possano sempre essere visti come se il primo fosse GR32. Ovviamente, se lo stack si dimostra non sufficiente, il suo contenuto deve essere preventivamente salvato in memoria e ristabilito in seguito.

[2] Si veda il Paragrafo 10.2.3 del libro sulla ridenominazione dei registri

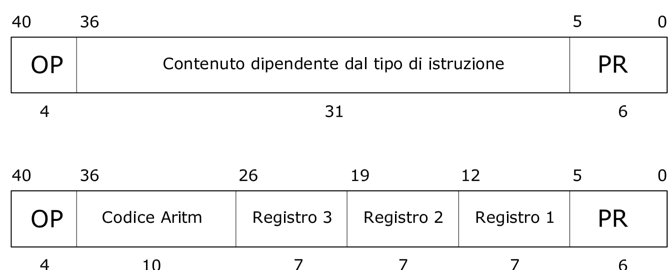


Figura D.3 Esempio di un formato istruzioni. Sostanzialmente l'Itanium una macchina RISC. Gli unici campi fissi sono OP e PR. La figura in basso mostra una operazione tra registri; tre campi individuano i registri, mentre i 10 bit restanti specificano esattamente qual è l'operazione. A seconda dell'istruzione i campi possono presentare più sottocampi all'interno.

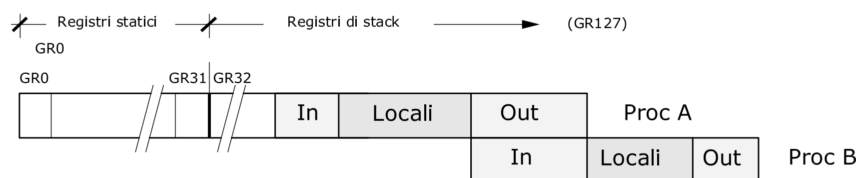


Figura D.4 Schematizzazione della chiamata/ritorno alle/dalle procedure.

Resta pure compito della procedura chiamata ristabilire eventuali registri statici che fossero stati modificati, ma devono risultare immutati al ritorno.

D.1.2 Istruzioni predicative

Nelle architetture tradizionali (praticamente tutte quelle menzionate nel libro e nelle appendici, con l'eccezione dell'architettura ARM), l'esecuzione condizionale è ottenuta attraverso salti condizionali, ovvero diramazioni (*branch*). Esse sono una delle maggiori cause di degradazione delle prestazioni. Per ridurre l'effetto delle previsioni errate, l'Itanium prevede l'esecuzione di istruzioni predicative, ovvero istruzioni che hanno effetto solo se è vero un predicato (condizione) ad esse associato. Appoggiandosi a questa tecnica, il compilatore è in grado di ridurre fortemente le diramazioni e, conseguentemente, le relative penalizzazioni in caso di errata previsione. Per esempio, il compilatore può allocare i due rami di un `if-then-else` a differenti unità funzionali e farle procedere in parallelo prendendo il risultato solo dal percorso corrispondente al verificarsi di una condizione. In tal modo è possibile eseguire ambedue i lati dell'`if` nel medesimo tempo che corrisponde all'esecuzione di uno solo.

Per capire le istruzioni predicative e come queste possono eliminare le diramazioni, si faccia riferimento al seguente pseudo codice.

```
if(p)  r1= r2+r3
```

Con una architettura convenzionale la precedente istruzione comporta comunque una istruzione di salto condizionato.

Alternativamente, non ci vuole molto a immaginare che la macchina effettui comunque la somma, ma modifichi lo stato di macchina (assegni il valore a `r1` e modifichi eventualmente i bit di stato influenzati dall'operazione, come ad esempio il bit di riporto) solo se il predicato `p` è vero, altrimenti non modifichi lo stato di macchina, in pratica comportandosi come una istruzione `nop`.

Il valore al predicato può essere assegnato attraverso l'istruzione *compare* (`cmp`), con la quale viene valutata una condizione e assegnato il valore vero (T/1) e falso (F/0) a uno o due registri predicativi. Consideriamo questo tratto di codice C

```
if (a==0)
   b= b+1;
else
   b= b-1;
```

Il salto condizionale per `a==0` può essere evitato se la macchina esegue in modo predicativo. In pseudo codice, riscrivendo il tratto precedente come

```
pT, pF = compare(a,0)
if (pT) b= b+1
if (pF) b= b-1
```

dove il predicato `pT` viene messo a 1 se la condizione `a==0` è vera, viene messo a 0 se è falsa. Il predicato `pF` è il complemento di `pT`.

Si noti che il salto è eliminato, poiché la dipendenza di controllo è stata convertita in una dipendenza dati, quella delle due istruzioni `b= b+1` e `b= b-1` dal `compare(a,0)`. Le due istruzioni predicate da `pT` e `pF`, essendo indipendenti tra di loro, possono essere compilate in modo da eseguire in parallelo nello stesso gruppo, nello stesso periodo di clock, con il risultato che solo l'istruzione il cui predicato è verificato produce effetto.

Vogliamo ora mostrare come quanto sopra trova realizzazione con l'Itanium. A tale scopo, prima di procedere spendiamo due parole sul suo linguaggio assembler, limitandoci al genere di statement di nostro interesse il cui formato è il seguente (tra parentesi quadre le parti opzionali).

```
[(qp)] mnemonic[.completers] dests=sources [;;]  //comments
```

Dove:

`(qp)` – nome di un registro predicativo; deve essere racchiuso tra parentesi. Se al tempo di esecuzione il registro predicativo `qp` contiene 1, allora l'istruzione viene eseguita e produce i suoi effetti; se contiene 0, allora l'istruzione viene comunque eseguita, ma non produce effetti. Se il registro predicativo non viene specificato, la logica della CPU assume che sia `p0`[3].

`mnemonic.completers` – nome simbolico dell'istruzione; possono essere impiegati uno o più postfissi, che completano il significato dell'istruzione;

`dests=srcs` – rappresentano gli operandi di destinazione e sorgente; alcune istruzioni possono avere due operandi destinazione e uno o più operandi sorgente; gli operandi multipli sono separati da virgola; quando tutti gli operandi sono destinazione o sorgente il segno "=" viene omesso;

[3] Notare che nel linguaggio assembler i registri generali vengono indicati come `r0-r127`, quelli in virgola mobile come `f0-f127`, quelli predicativi come `p0-p63`.

`;;` – marca esplicitamente uno stop; esso stabilisce che l'istruzione è l'ultima di un gruppo (e può essere eseguita in parallelo alle eventuali istruzioni che la precedono); può apparire a seguito dell'istruzione o come statement individuale successivo;
`//` – quel che segue è un commento.

Ad esempio

```
     cmp.ne p1,p0=r4,0
     ;;
(p1) add r1=r2,r3
```

La prima istruzione confronta il contenuto del registro di uso generale r4 (sorgente) con 0; se la condizione è vera (cioè, se il contenuto di r4 è diverso da 0) allora a p1 viene assegnato il valore 1, a p0 il valore 0. Il doppio punto e virgola determina l'inserimento di uno stop a seguire l'istruzione. La seconda istruzione fa la somma di r2 con r3 e lo assegna a r1, se e solo se il registro p1 contiene 1, ovvero se il precedente confronto ha dato risultato positivo.

L'impiego di un predicato richiede uno stop per separare l'istruzione che lo produce da quelle che lo usano. L'eccezione a questa regola si ha quando un confronto tra interi assegna un valore a un predicato usato nella successiva istruzione di salto condizionato. Ad esempio

```
     cmp.eq p1,p2=r1,r2    //Stop non richiesto
(p1) br.cond dest
```

dove la seconda riga ha effetto (saltare a `dest` se `cond` è vera) solo se `p1` è a 1.

Torniamo ora al nostro statement C.

```
if (a==0)
   b= b+1;
else
   b= b-1;
```

esso diventa

```
     comp.eq p1,p2 = a,0 ;;
(p1) add b= b,1         //if a==0 then add
(p2) sub b= b,1         //if a!=0 then sub
```

Poiché le istruzioni sui due percorsi dell'`if` sono predicate da condizioni complementari, è garantito che esse non confliggono. Generalizzando, il compilatore può accorpare le istruzioni nello stesso gruppo in modo da fare uso ottimale delle risorse.

Confrontiamo il precedente codice con quello che si avrebbe con una macchina con architettura ×86.

```
       cmp  a,0
       jne  lab1
       add  b,1
       jmp  fine
lab1:  sub  b,1
fine:  :::
```

L'uso del predicato ha consentito di eliminare il salto condizionato. La riduzione delle istruzioni di salto condizionato riduce la probabilità di dover svuotare la pipeline. Inoltre, l'impaccamento delle istruzioni nello stesso gruppo, riduce il numero di cicli di clock richiesto. Nel caso dell'Itanium sono richiesti 2 cicli di clock, nel caso $\times 86$ i cicli sono 4 se il confronto dà esito positivo, 3 se dà esito negativo. (Per semplicità si è assunto che ogni istruzione esegua in un ciclo.)

Il seguente tratto di codice mostra come viene costruito un salto "multivie". Tre salti condizionati sono nello stesso pacchetto (pacchetto di tipo BBB); se nessuno ha effetto il controllo prosegue con il gruppo successivo.

```
             ...  //pacchetto conclusivo di un gruppo
             (p1) br.cond1 label_B
             (p2) br.cond2 label_C
             (p3) br.cond3 label_D
                  //gruppo successivo
label_A:     ...
```

Supponendo che solo il predicato `p2` si vero, il salto a `label_C` ha effetto solo se è verificata `cond2`. L'ordine dei salti nel pacchetto è importante, a meno che i salti non siano mutuamente esclusivi. Se questo non è il caso ha effetto il primo, nell'ordine testuale, che verifichi predicato e condizione.

D.1.3 Esecuzione speculativa

Come sappiamo l'esecuzione speculativa cerca di ridurre la latenza dovuta a certe operazioni anticipandone l'esecuzione rispetto a quello che sarebbe il normale flusso di una macchina puramente sequenziale, salvo verificare a posteriori se la speculazione è confermata.

Speculazione sul flusso di controllo

L'architettura EPIC riduce gli effetti delle diramazioni eliminandole con l'esecuzione speculativa. Tuttavia non tutte le diramazioni possono essere eliminate.

Nella architettura EPIC la speculazione è riferita oltre che ai salti alle operazioni di load. Queste ultime sono quelle che hanno la massima latenza, condizionando tutte le istruzioni seguenti che fanno uso dei dati letti. Un modo per render più efficiente l'esecuzione consiste nell'anticipare le istruzioni di load, in modo da annullare la latenza, ma in tal caso è poi necessario verificare se l'aver modificato la posizione del load non ha indotto errori. L'architetture fornisce il supporto per tale verifica.

A tale scopo, è prevista l'istruzione di "load speculativo" che dal punto di vista sintattico assume la forma `ldx.s`, dove `x` può assumere i valori, 1, 2, 4 o 8[4], a seconda che debba essere caricato 1 byte, ovvero 2, 4 o 8 rispettivamente. Il postfisso `s` (qualificatore) indica che questa è una load speculativa, distinguendola da una normale load.

Si consideri il seguente codice Itanium scritto in modo "sequenziale"[5].

```
       cmp.eq p1, p0 = r10, 10      // clock 0
(p1)   br.cond end_if ;;            // clock 0
```

[4]Ovviamente `ld8` determina il caricamento nel registro di destinazione di una parola di 64 bit.

[5]Nel codice ci sono 3 gruppi di istruzioni e come minimo ci vogliono 3 clock; qui diventano 4 assumendo che la load richieda 2 clock.

```
        ld8 r1 = [r2] ;;                // clock 1 e 2
        add r3 = r1, r4                 // clock 3
end_if: ...
```

Il compilatore potrebbe riaggiustare la posizione del load facendolo diventare speculativo (`ld8.s`), portandolo a precedere di almeno 2 clock il resto delle istruzioni.

```
        ld8.s r1 = [r2] ;;              // clock -2 (almeno)
        ....
        cmp.eq p1, p0 = r10, 10         // clock 0
(p1)    br.cond end_if                  // clock 0
        chk.s r1, ripara ;;             // clock 0
        add r3 = r1, r4                 // clock 1
end_if: ...
           ...
ripara:....
```

L'istruzione `ld8.s` potrebbe generare un'eccezione (perché per esempio r2 contiene 0). In tal caso il trattamento dell'eccezione viene ritardato. Quel che accade è che `ld8.s` asserisce il bit NaT associato a r1. Più avanti, quando verrà eseguito il gruppo di istruzioni al clock 0, verrà verificato se la diramazione andava presa e/o se c'è stata un'eccezione (istruzione `chk.s`, *check speculation*). Se c'è stata un'eccezione, il controllo passa all'istruzione in `ripara`, dove può essere previsto il codice per rimediare, compresa la riesecuzione del load; se invece non c'è stata eccezione, e la diramazione è effettiva, si attua il salto, ma la logica di macchina (dall'accoppiata `br chk.s r1`) assicura che il risultato del load speculativo non abbia l'effetto di modificare r1.

Se il risultato di due o più load speculativi viene usato nel medesimo calcolo, gli eventuali NaT possono diventare due o più. Il compilatore può inserire un solo `chk.s` alla fine, verificando in un solo colpo il risultato del calcolo. Ovvero, l'eventuale eccezione è ritardata al momento in cui tutti i load speculativi dello stesso calcolo sono stati effettuati.

Speculazione sui dati

Al Paragrafo 10.2.4 del libro è stato osservato che gli indirizzi di memoria non sempre possono essere calcolati al tempo di compilazione. Un linguaggio come il C usa il puntatori per accedere alla memoria e ciò rende impossibile per il compilatore stabilire quale posizione viene indirizzata. Al tempo di esecuzione possono sorgere conflitti, come nel caso delle due istruzioni che seguono, dove r1 e r4 potrebbero indirizzare la stessa posizione di memoria.

```
st8  [r1] = r2
ld8  r3 = [r4]
add r7= r7, r3
```

Anche per questo caso esiste una sorta di load speculativo, detto *advance load* (`ld.a`). Esso può essere posizionato in modo da iniziare il caricamento prima che il dato serva, verificando successivamente, prima di usare il dato, tramite l'istruzione *load check* (`ldx.c`), se c'è un conflitto con uno store. La sequenza precedente verrebbe compilata come quella che segue.

```
ld8.a  r3 = [r4]
     . . .     ;;
st8    [r1] = r2
```

```
ld8.c  r3 = [r4]
add    r7 = r7, r3
```

Il load anticipato avvia il caricamento sufficientemente prima che il dato che esso legge venga usato. Se lo store (st8) fa riferimento alla stessa posizione, allora il dato letto è da scartare. L'istruzione `ld8.c`, a seguire la `st8` fa la verifica della coincidenza degli indirizzi e, se del caso, riesegue automaticamente l'istruzione di load.

Nel caso in cui le cose procedano come da speculazione, la latenza dovuta al load risulta eliminata.

D.1.4 Conclusioni

L'Itanium, almeno sulla carta, è una eccellente architettura. Sostanzialmente si tratta di una macchina RISC, che lascia al compilatore il carico di organizzare le istruzioni in modo da fare l'uso più efficiente delle unità funzionali.

L'Itanium avrebbe dovuto essere il prototipo di una nuova generazione di calcolatori a 64 bit e per essa venne introdotta la sigla IA-64, con la quale si ammiccava al superamento della precedente architettura IA-32, ovvero la $\times 86$ a 32 bit. Con l'Itanium non sono stati raggiunti gli obiettivi che il costruttore si era proposto. A seguito dell'estensione a 64 bit dell'architettura $\times 86$ da parte di AMD, anche Intel ha dovuto incamminarsi su questa strada, ridimensionando la portata del progetto Itanium. Indubbiamente, le condizioni di mercato, la grande diffusione delle macchine $\times 86$ e l'immenso patrimonio di software per esse sviluppato sono stati i fattori determinanti nel non consentire all'Itanium di emergere.

D.2 Siti web

Per l'Itanium si deve cercare sui siti Intel. Sono disponibili svariati manuali; all'indirizzo `http://www.intel.com/design/itanium2/manuals/251110.htm` si trova il *Processor Reference Manual for Software Development and Optimization* dell'Itanium 2.

Bibliografia

[Dul98] C. Dulong, *The IA-64 architecture at work*, IEEE Computer **31** (1998), no. 7, 24–32.

[Fis83] J. A. Fisher, *Very long instruction word architectures and the ELI-512*, Proc. of ISCA '83, 10th Int'l Symp. Computer Architecture, Giugno 1983, pp. 140–150.

[HMR+00] J. Huck, D. Morris, J. Ross, A. Knies, H. Mulder, and R. Zahir, *Introducing the IA-64 architecture*, IEEE Micro **20** (2000), no. 5, 12–23.

[LBT06] J. Liu, B. Bell, and T. Truong, *Analysis and characterization of Intel Itanium instruction bundles for improving VLIW processor performance*, Proc. of the First International Multi-Symposiums on Computer and Computational Sciences (IMSCCS'06) (Hangzhou, China), Giugno 2006.

[MS03] C. McNairy and D. Soltis, *Itanium 2 processor microarchitecture*, IEEE Micro (2003), 44–55.

[RGB+02] R. Rakvic, E. Grochowski, B. Black, M. Annavaram, T. Diep, , and J. P. Shen, *Performance advantage of the register stack in intel Itanium processors*, Workshop on Explicit Parallel Instruction Computing (EPIC) Architectures and Compiler Techniques, 2002.

[RMC04] S. Rusu, H. Muljono, and B. Cherkauer, *Itanium 2 processor 6m: higher frequency and larger l3 cache*, IEEE Micro (2004), 10–18.

[ZRH00] R. Zahirn, J. Ross, and D. Morrisand D. Hess, *Os and compiler considerations in the design of the IA-64 architecture*, Proceedings of ASPLOS-IX, the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (Cambridge, MA), Novembre 2000.

Indice analitico

Indice